---

title: "How to Cache Exchange Rates and Avoid Rate Limit Errors" published: true tags: api, javascript, python, tutorial canonical_url: https://allratestoday.com/blog/how-to-cache-exchange-rates-avoid-rate-limits/ cover_image:

You integrated an exchange rate API, deployed to production, and then hit HTTP 429: `Rate limit exceeded`. This is the most common operational issue developers face with exchange rate APIs — and it is entirely preventable with proper caching.

Why You Need to Cache Exchange Rates

Exchange rates don't change every millisecond. Even "real-time" APIs like AllRatesToday update every 60 seconds. Fetching the same rate on every page load is wasteful and will quickly exhaust your quota.

Consider a simple e-commerce site with 10,000 daily visitors:

- **Without caching:** 10,000 API calls/day = 300,000/month. You'll need a Large plan.
- **With 5-minute cache:** 288 API calls/day = 8,640/month. A Small plan covers it.
- **With 1-hour cache:** 24 API calls/day = 720/month. The Free tier is enough.

> **The math:** If your API updates every 60 seconds, caching for 5 minutes means you are at most 4 minutes behind — which is perfectly acceptable for price displays and most applications.

Choosing Your Cache Duration (TTL)

Use Case	Recommended TTL	Why
Trading / Forex	No cache or 1–5s	Every second counts
Payment checkout	30–60 seconds	Rate should be current at time of charge
E-commerce price display	1–5 minutes	Visitors tolerate slight delays
Dashboard / analytics	5–15 minutes	Refresh rates match dashboard refresh
Invoicing / billing	1–24 hours	Rate is locked at invoice generation

Use Case	Recommended TTL	Why
Accounting reports	24 hours	End-of-day rates are standard

Strategy 1: Simple In-Memory Cache (JavaScript)

No dependencies needed:

```
import AllRatesToday from '@allratestoday/sdk';

const client = new AllRatesToday('YOUR_API_KEY');
const cache = new Map();
const CACHE_TTL = 5 * 60 * 1000; // 5 minutes

async function getRate(source, target) {
  const key = `${source}_${target}`;
  const cached = cache.get(key);

  if (cached && Date.now() - cached.timestamp < CACHE_TTL) {
    return cached.rate;
  }

  const rate = await client.getRate(source, target);
  cache.set(key, { rate, timestamp: Date.now() });
  return rate;
}

// Only hits the API once per 5 minutes per currency pair
const rate1 = await getRate('USD', 'EUR');  // API call
const rate2 = await getRate('USD', 'EUR');  // Served from cache
```

Strategy 2: Simple In-Memory Cache (Python)

```
import time
from allratestoday import AllRatesToday

client = AllRatesToday("YOUR_API_KEY")
_cache = {}
CACHE_TTL = 300  # 5 minutes

def get_rate(source: str, target: str) -> float:
    key = f"{source}_{target}"
    cached = _cache.get(key)

    if cached and time.time() - cached["timestamp"] < CACHE_TTL:
```

```
        return cached["rate"]

    rate = client.get_rate(source, target)
    _cache[key] = {"rate": rate, "timestamp": time.time()}
    return rate
```

Strategy 3: Stale-While-Revalidate

The best strategy for production. Serve cached (possibly stale) data immediately, refresh in the background. Users never wait for an API call.

```
import AllRatesToday from '@allratestoday/sdk';

const client = new AllRatesToday('YOUR_API_KEY');
const cache = new Map();
const FRESH_TTL = 60 * 1000;       // Fresh for 1 minute
const STALE_TTL = 10 * 60 * 1000;  // Serve stale up to 10 minutes
const refreshing = new Set();

async function getRate(source, target) {
  const key = `${source}_${target}`;
  const cached = cache.get(key);
  const now = Date.now();

  if (cached) {
    const age = now - cached.timestamp;

    if (age < FRESH_TTL) {
      return cached.rate; // Fresh
    }

    if (age < STALE_TTL) {
      // Stale but usable — refresh in background
      if (!refreshing.has(key)) {
        refreshing.add(key);
        client.getRate(source, target)
          .then(rate => cache.set(key, { rate, timestamp: Date.now() }))
          .finally(() => refreshing.delete(key));
      }
      return cached.rate;
    }
  }

  // No cache or expired
  const rate = await client.getRate(source, target);
  cache.set(key, { rate, timestamp: now });
  return rate;
}
```

> **Why this works:** Exchange rates don't jump dramatically in 10 minutes. Serving a 5-minute-old rate instantly is better than making your user wait 200ms for a rate that's 0.01% more accurate.

Strategy 4: Redis Cache (High-Traffic Apps)

For multi-instance applications, use Redis to share a cache:

```
import AllRatesToday from '@allratestoday/sdk';
import Redis from 'ioredis';

const client = new AllRatesToday('YOUR_API_KEY');
const redis = new Redis(process.env.REDIS_URL);
const CACHE_TTL = 300; // 5 minutes

async function getRate(source, target) {
  const key = `exchange_rate:${source}:${target}`;

  const cached = await redis.get(key);
  if (cached) return parseFloat(cached);

  const rate = await client.getRate(source, target);
  await redis.setex(key, CACHE_TTL, rate.toString());
  return rate;
}
```

Strategy 5: Batch Prefetch

If you know which currencies you need, fetch them all on a schedule:

```
import AllRatesToday from '@allratestoday/sdk';

const client = new AllRatesToday('YOUR_API_KEY');
let ratesCache = {};

const CURRENCIES = ['EUR', 'GBP', 'JPY', 'CAD', 'AUD', 'CHF', 'INR'];

async function refreshAllRates() {
  try {
    const allRates = await client.getRates('USD');
    const rates = {};
    for (const currency of CURRENCIES) {
      rates[`USD_${currency}`] = allRates[currency];
```

```
    }
    rates.lastUpdated = Date.now();
    ratesCache = rates;
  } catch (err) {
    console.error('Refresh failed, keeping stale cache:', err.message);
  }
}

refreshAllRates();
setInterval(refreshAllRates, 5 * 60 * 1000);

// Synchronous access — no await needed
function getRate(source, target) {
  return ratesCache[`${source}_${target}`];
}
```

> **Request math:** 1 API call per 5 minutes = 288 calls/day = 8,640/month. Serves unlimited concurrent users.

Common Mistakes to Avoid

- **Fetching on every request:** The #1 cause of rate limit errors. Always cache.
- **Client-side API calls:** Never expose your API key in frontend JavaScript. Fetch server-side and proxy.
- **Cache per user:** Exchange rates are the same for everyone. Use a global cache.
- **Ignoring cache stampede:** When cache expires, 100 concurrent requests all hit the API. Use stale-while-revalidate or a lock.

> **Security:** Never call the exchange rate API directly from the browser. Your API key will be visible in the network tab. Always proxy through your backend.

Summary

Strategy	Best For	Complexity
Simple TTL cache	Small apps, single server	Low
Stale-while-revalidate	User-facing apps needing instant response	Medium

Strategy	Best For	Complexity
Redis cache	Multi-instance / high-traffic	Medium
Batch prefetch	Known currency set, predictable traffic	Low

AllRatesToday — 60-second updates, 160+ currencies, official SDKs. Start with the free tier — plenty for a cached application.

Get your free API key | Documentation | GitHub